CS 4/591: Neural Network
Assignment 1

Jyrus Cadman Shahriar Dipon Robert McCourt Bethany Pena
Gabriel Urbaitis

17 September 2024

Contents

1 Introduction

The perceptron is the simplest neural network, consisting of a single input layer and output node. The perceptron is a linear classifier, capable of constructing a hyperplane through any set of linear separable data. As one of the building blocks of more advanced neural network architectures, it is a fundamental model to explore.

This report explores the implementation and testing of the perceptron, its learning algorithm, and the gradient descent optimization. The Perceptron Learning Algorithm updates the model's weights iteratively based on misclassified samples. Gradient Descent, a widely used optimization algorithm, searches for the set of weights that minimize a loss function. The Gradient Descent Algorithm does this by iteratively updating the weights in the direction of the negative gradient of the cost function until it discovers its local minimum.

Our implementation is tested on three different sets of randomly generated data which are then filtered into two classes, one greater and one less than a decision boundary specified for each case in the project requirements. The filtered samples are accumulated until there are 100 samples for each class in both the training and test sets.

We trained our model using both algorithms and using two different learning rates. We then compared the accuracy and explain the behavior we observe in our testing.

The goal of this work is to deepen our understanding of how these algorithms work through implementing them in Python code and testing their performance on randomly generated linearly separable data.

2 Implementation

2.1 The Perceptron Class

A perceptron is a simple neural network model with an input layer, an activation function, and an output node. The input layer has n nodes that pass n features to the output node. The output node calculates a linear combination of the inputs and their weights, then applies the activation function, a sign function, to produce a prediction.

The perceptron model can be implemented in Python by defining a Perceptron class. The perceptron class contains the following methods and attributes:

Attributes:

weights: an array containing the weights to be learned by the perceptron; they are randomly initialized before training.

bias: a float that is randomly initialized before training.

learning rate: the learning rate can be set by the user, otherwise is defaulted to 0.01.

Methods:

forward: computes the forward pass of the Perceptron; this is done by computing $\mathbf{w}^T\mathbf{X} + b$.

predict: applies the sign activation function to predict the class labels of the input samples based on the forward pass.

fit: finds the optimal weights according to the Perceptron Learning Algorithm.

2.2 Perceptron Learning Algorithm

The Perceptron Learning Algorithm aims to classify the data over multiple iterations or epochs. The algorithm aims to learn a hyperplane which will linearly separate the data into different classes. In the initial epochs, the perceptron will make the most errors. For data instances the perceptron fails to classify, the perceptron algorithm updates its weight based to correctly identify the misclassified label in a later epoch during training. The algorithm uses the error to "move" the hyperplane towards the correct classification. For example, if for some training sample, the correct label is $y = 1$ and the predicted label$\hat{y} = -1$, the error will be positive and the weights will be updated so that they move the hyperplane in such a way that this sample will be predicted positively.

Similarly, the update for bias takes place. The bias adjusts the distance of the hyperplane from the origin. Because of this, it is only updated using the error. It should be noted that the weights and bias updates only occur for data instances which are misclassified.

The following expression defines the iterative process for updating the weights and biases according to the Perceptron Learning Algorithm. The learning rate is a hyper-parameter given by α and defines how large a "step" we should in updating the weight vector.

$$\mathbf{W} \leftarrow \mathbf{W} + \alpha(y - \hat{y})\mathbf{X}^{\mathbf{T}}$$
$$b \leftarrow b + \alpha(y - \hat{y})$$

The Perceptron Learning Algorithm is most effective when the data is linearly separable because it depends on discovering the parameters of a hyperplane to separate the data into two classes. This can be beneficial if the data is linearly separable because we are guaranteed convergence. However, if the data is not linearly separable, it may not converge. Typically our data will not be linearly separable, so more robust methods for classification should be explored.

2.2.1 Perceptron Learning Algorithm Implementation

We can implement the Perceptron Learning Algorithm in Python as a `fit` method in the Perceptron class. The Python implementation follows the above algorithm by iterating through the training samples, obtaining a prediction for each sample, and updating the weight values according to the error, inputs, and learning rate. The bias is also updated according to the error and learning rate. The algorithm terminates when all classifications are correct, or the maximum number of epochs are reached.

2.3 Gradient Descent Algorithm

If we are given a loss function, such as the mean squared error, that describes the error in our predictions, we want to find the weights for our neural network which minimize the loss. Gradient Descent is an iterative optimization technique used to discover the set of weights that minimizes the loss function.

For our perceptron, we use the mean squared error (MSE) as our loss function.

$$MSE = \frac{1}{N}\sum_{n=0}^{N}(y - \hat{y})^2$$

The Gradient Descent Algorithm updates the weights in the negative direction of the gradient of the loss function with respect to the weights. This means it will iteratively step towards the local minimum of the loss function. The Gradient Descent Algorithm is given by the following expression. The function $\mathcal{L}$ defines the loss function.

$$\begin{aligned}
&\mathbf{W} \leftarrow \mathbf{W} + \alpha \nabla \mathcal{L}(\mathbf{w}, \mathbf{X}, \mathbf{y}) \\
&b \leftarrow b + \alpha y
\end{aligned}$$

An advantage of using the Gradient Descent Algorithm is that is can be used with non-linearly separable data and we can achieve convergence. The stability of its convergence depends on setting the learning rate sufficiently small that it can find the local or global minimum. One challenge with the Gradient Descent Algorithm is, depending on how the model's weights are initialized, it may find a local minimum rather than a global minimum.

2.3.1 Gradient Descent Algorithm Implementation

We implemented the Gradient Descent Algorithm in Python as a `fit_GD` method in the Perceptron class. Inside `fit_GD` and at the beginning of each epoch, the mean squared error for the training set is calculated. For data instances that are misclassified, the weights of the perceptron algorithm are updated in the negative direction of the loss function. This process is repeated across multiple epochs until the misclassifed data instances are correctly classified. The number of epochs that will run inside `fit_GD` is a hyperparameter the user sets. However, if all the data instances are correctly classified before reaching the last epoch, the training stops.

3 Testing

3.1 Testing Data

We tested our algorithms using three different cases of training and testing data. Each training and test data set contained data that is classified into two classes, each class containing 100 samples each.

Case 1 :

Class 1: $\{(x_1, x_2)| - x_1 + x_2 > 0\}$

Class 2: $\{(x_1, x_2)| - x_1 + x_2 < 0\}$

Case 2 :

Class 1: $\{(x_1, x_2)|x_1 - 2x_2 + 5 > 0\}$

Class 2: $\{(x_1, x_2)|x_1 - 2x_2 + 5 < 0\}$

Case 3 :

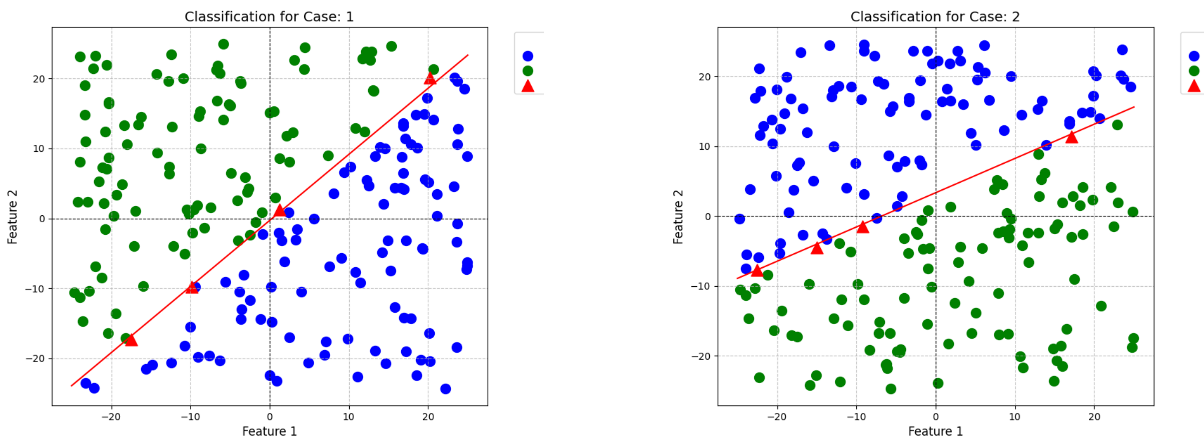
Class 1: $\{(x_1, x_2, x_3, x_4)|0.5x_1 - x_2 - 10x_3 + x_4 + 50 > 0\}$

Class 2: $\{(x_1, x_2, x_3, x_4)|0.5x_1 - x_2 - 10x_3 + x_4 + 50 < 0\}$

The data was generated by randomly sampling points from a uniform distribution that were consistent with the class definitions for each case. We used different random seeds for sampling the training and testing data to ensure we do not test and train the same data.

3.1.1 Training and Evaluating

As the learning rate is the only hyperparameter for binary class perceptrons, we tested two different learning rates, .01 and .001. For each dataset, we trained both using the Perceptron Learning Algorithm and the Gradient Descent Algorithm. The accuracy results from our training can be seen in figure 2. In figure 1, we can visualize the classification results for two different cases, learning rates, and algorithms.



(a) Case 1: Perceptron Learning Algorithm (Learning Rate = 0.01)

(b) Case 2: Gradient Descent (Learning Rate = 0.001)

Figure 1: Classification Results Examples

The Perceptron Learning Algorithm performed markedly better on the data generated for Case 3, classifying all samples correctly using both learning rates. This is compared to 88 percent for a .001 learning rate and 87.5 percent for a .01 learning rate using the Gradient Descent Algorithm.

We found that the random seeds we chose had a greater effect on the accuracy than the testing method, especially when using the larger .01 learning rate. We found that when using a larger learning rate, the accuracy for our model trained with Gradient Descent were below 25 percent, suggesting that in those cases that the algorithm may have been stuck in a local minima.

4 Discussion

In our testing, we would expect the linearly separable data to converge to 100 percent accuracy, however, we do see this for all test cases in our testing results. We believe that

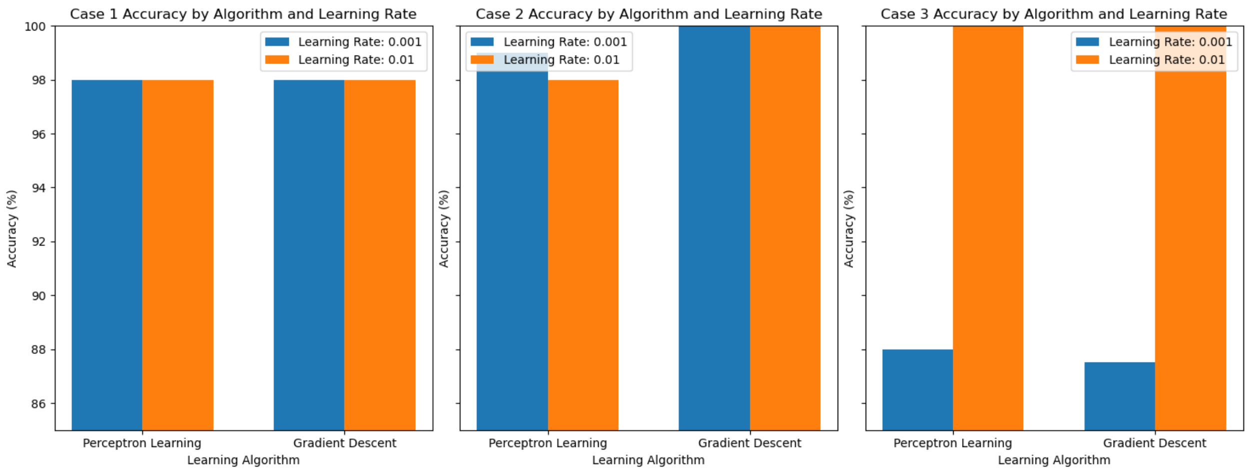


Figure 2: Accuracy by Algorithm and Learning Rate for Different Datasets

this indicates further testing to find correct learning rate would be needed. If we had more time, we would attempt adjust the learning rate in response to our intermediate data on the loss each epoch.

5 Conclusion

This work explored the implementation and training of the Perceptron neural network using the Perceptron Learning Algorithm and the Gradient Descent Algorithm. We trained our Perceptron using three different cases of randomly generated data and two different learning rates. Although we did not obtain the performance we expected for the Gradient Descent Algorithm, we hypothesized why we think we didn't obtain the performance we expected, and what we would explore in the future to potentially improve our results.

Appendix

```python
import sys

import numpy as np

class Perceptron:
    """
    A Perceptron classifier.

    This class implements both the original Perceptron learning algorithm
    [Rosenblatt 1958] and a variant using the gradient descent optimizaiton.

    Attributes
    ----------
    weights : numpy.ndarray
        The weight vector of the Perceptron.
    bias : float
        The bias term of the Perceptron.
    """

    def __init__(self, num_inputs, learning_rate):
        """
        Initialize the Perceptron with random weights and a bias, and set
        the learning rate.

        Parameters
        ----------
        num_inputs : int
            The number of input features.
        learning_rate : float
            The learning rate (default is 0.01).
        """
        self.weights = np.random.uniform(-1, 1, num_inputs)
        self.bias = np.random.uniform(-1, 1)
```

```
        self.learning_rate = learning_rate

    def forward(self, inputs):
        """
        Compute the forward pass of the Perceptron.

        Parameters
        ----------
        inputs : numpy.ndarray
            The input samples.

        Returns
        -------
        numpy.ndarray
            The output of the Perceptron before thresholding.
        """
        return np.dot(inputs, self.weights) + self.bias

    def predict(self, inputs):
        """
        Predict the class labels for the input samples.

        Parameters
        ----------
        inputs : numpy.ndarray
            The input samples.

        Returns
        -------
        numpy.ndarray
            The predicted class labels (-1 or 1).
        """
        return np.where(self.forward(inputs) >= 0, 1, -1)

    def fit(self, data, labels, max_epochs=10000):
        """
        Fit the Perceptron to the training data using the
           original algorithm.
```

```

        Parameters
        ----------
        data : numpy.ndarray
            The training samples.
        labels : numpy.ndarray
            The target values.
        max_epochs : int, optional
            The maximum number of epochs. Defaults to 100.
        """
        for epoch in range(max_epochs):
            all_correct = True # Assume all predictions will be correct at the start of each epoch

            for inputs, label in zip(data, labels):
                prediction = self.predict(inputs)
                error = label - prediction

                if error != 0:
                    all_correct = False # Set to False if any prediction is incorrect

                    # Update the weights and bias based on the error
                    update = self.learning_rate * error
                    self.weights += update * inputs
                    self.bias += update

            if all_correct:
                print(f"All predictions correct after {epoch + 1} epochs.")
                return

        print(f"Reached max_epochs ({max_epochs}).")

        """
        Get to a point where once the loss plateaus, or doesn't get any better, exit at that epoch and return it
```

```
        """

    def fit_GD(self, data, labels, max_epochs=10000, error_threshold=0.001, patience=50):
        """
        Train the perceptron using gradient descent and stop early if the loss plateaus.

        Parameters
        ----------
        data : np.ndarray
            Training data.
        labels : np.ndarray
            Training labels.
        max_epochs : int, optional
            Maximum number of epochs. The default is 10000.
        error_threshold : float, optional
            Threshold for the change in loss to consider it as plateaued. The default is 0.001.
        patience : int, optional
            Number of consecutive epochs to wait for improvement before stopping. The default is 10.

        Returns
        -------
        None.
        """
        num_samples, num_features = data.shape
        best_loss = float('inf')
        epochs_without_improvement = 0

        for epoch in range(max_epochs):
            predictions = self.predict(data)

            errors = labels - predictions

            # Mean Squared Error (MSE) Loss (variable used to assess convergence)
```

```
            mse_loss = ((1 / num_samples) * np.sum(errors ** 2)) * 100

            # Gradient for weights
            dw = (-2 / num_samples) * np.dot(data.T, errors)
            # Gradient for bias
            db = (-2 / num_samples) * np.sum(errors)

            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

            # Check if predictions are correct
            all_correct = np.all(errors == 0)

            # Continually check convergence every 10 epochs
            if epoch % 10 == 0:
                print(f"Epoch {epoch}/{max_epochs}, MSE Loss: {mse_loss:.5f}")

            # Check for early stopping
            if mse_loss < best_loss - error_threshold:
                best_loss = mse_loss
                epochs_without_improvement = 0
            else:
                epochs_without_improvement += 1

            if epochs_without_improvement >= patience:
                print(f"Loss plateaued after {epoch + 1} epochs.")
                return

            if all_correct:
                print(f"All predictions correct after {epoch + 1} epochs.")
                return

        print(f"Reached max_epochs ({max_epochs}).")
```

Listing 1: perceptron.py

```python
import numpy as np
import pandas as pd

def generate_case1_data():
    np.random.seed(24)
    case1_class1 = []
    while len(case1_class1) < 100:
        samples = np.random.uniform(-25, 25, (100, 2))
        # Class 1: -x1 + x2 > 0
        filtered_samples = samples[samples[:, 1] > samples[:, 0]]
        case1_class1.extend(filtered_samples)

    case1_class1 = np.array(case1_class1[:100])

    case1_class2 = []
    while len(case1_class2) < 100:
        samples = np.random.uniform(-25, 25, (100, 2))
        # Class 2: -x1 + x2 < 0
        filtered_samples = samples[samples[:, 1] < samples[:, 0]]
        case1_class2.extend(filtered_samples)

    case1_class2 = np.array(case1_class2[:100])

    X_train = np.vstack((case1_class1, case1_class2))
    y_train = np.hstack((np.ones(len(case1_class1)), -1 * np.ones(len(case1_class2))))

    # different seed for testing data
    np.random.seed(25)

    case1_class1_test = []
    while len(case1_class1_test) < 100:
        samples = np.random.uniform(-25, 25, (100, 2))
        filtered_samples = samples[samples[:, 1] > samples[:, 0]]
        case1_class1_test.extend(filtered_samples)
```

```

    case1_class1_test = np.array(case1_class1_test[:100])

    case1_class2_test = []
    while len(case1_class2_test) < 100:
        samples = np.random.uniform(-25, 25, (100, 2))
        filtered_samples = samples[samples[:, 1] < samples[:,
            0]]
        case1_class2_test.extend(filtered_samples)

    case1_class2_test = np.array(case1_class2_test[:100])

    X_test = np.vstack((case1_class1_test, case1_class2_test))
    y_test = np.hstack((np.ones(len(case1_class1_test)), -1 * np
        .ones(len(case1_class2_test))))

    return X_train, y_train, X_test, y_test


def generate_case2_data():
    np.random.seed(24)
    case2_class1 = []
    while len(case2_class1) < 100:
        samples = np.random.uniform(-25, 25, (100, 2))
        # Class 1: x1 - 2x2 + 5 > 0
        filtered_samples = samples[samples[:, 0] - 2 * samples
            [:, 1] + 5 > 0]
        case2_class1.extend(filtered_samples)

    case2_class1 = np.array(case2_class1[:100])

    case2_class2 = []
    while len(case2_class2) < 100:
        samples = np.random.uniform(-25, 25, (100, 2))
        # Class 2: x1 - 2x2 + 5 < 0
        filtered_samples = samples[samples[:, 0] - 2 * samples
            [:, 1] + 5 < 0]
        case2_class2.extend(filtered_samples)
```

```

    case2_class2 = np.array(case2_class2[:100])

    X_train = np.vstack((case2_class1, case2_class2))
    y_train = np.hstack((np.ones(len(case2_class1)), -1 * np.
       ones(len(case2_class2))))

    #different seed for testing data
    np.random.seed(25)
    case2_class1_test = []
    while len(case2_class1_test) < 100:
        samples = np.random.uniform(-25, 25, (100, 2))
        filtered_samples = samples[samples[:, 0] - 2 * samples
           [:, 1] + 5 > 0]
        case2_class1_test.extend(filtered_samples)

    case2_class1_test = np.array(case2_class1_test[:100])

    case2_class2_test = []
    while len(case2_class2_test) < 100:
        samples = np.random.uniform(-25, 25, (100, 2))
        filtered_samples = samples[samples[:, 0] - 2 * samples
           [:, 1] + 5 < 0]
        case2_class2_test.extend(filtered_samples)

    case2_class2_test = np.array(case2_class2_test[:100])

    X_test = np.vstack((case2_class1_test, case2_class2_test))
    y_test = np.hstack((np.ones(len(case2_class1_test)), -1 * np
       .ones(len(case2_class2_test))))

    return X_train, y_train, X_test, y_test

def to_csv(X_train, y_train, X_test, y_test, case_number):
    train_df = pd.DataFrame(X_train, columns=['x1', 'x2'])
    train_df['label'] = y_train

    test_df = pd.DataFrame(X_test, columns=['x1', 'x2'])
```

```
    test_df['label'] = y_test

    train_df.to_csv(f'case{case_number}_train.csv', index=False)
    test_df.to_csv(f'case{case_number}_test.csv', index=False)

def main():
    # Case 1
    X_train_1, y_train_1, X_test_1, y_test_1 = generate_case1_data()
    to_csv(X_train_1, y_train_1, X_test_1, y_test_1, case_number=1)

    # Case 2
    X_train_2, y_train_2, X_test_2, y_test_2 = generate_case2_data()
    to_csv(X_train_2, y_train_2, X_test_2, y_test_2, case_number=2)

if __name__ == "__main__":
    main()

def generate_case3_data():
    # Generate 100 samples for Class 1
    np.random.seed(24)
    X_class1 = []
    while len(X_class1) < 100:
        samples = np.random.uniform(-25, 25, (100, 4))
        # Class 1: {(x1, x2, x3, x4) | 0.5x1   x2   10x3 + x4 + 50 > 0}
        filtered_samples = samples[0.5 * samples[:, 0] - samples[:, 1] - 10 * samples[:, 2] + samples[:, 3] + 50 > 0]
        X_class1.extend(filtered_samples)

    X_class1 = np.array(X_class1[:100])

    # Generate 100 samples for Class 2
    X_class2 = []
    while len(X_class2) < 100:
```

```
        samples = np.random.uniform(-25, 25, (100, 4))
        # Class 2: {(x1, x2, x3, x4) | 0.5x1    x2    10x3 + x4 + 50 < 0}
        filtered_samples = samples[0.5 * samples[:, 0] - samples[:, 1] - 10 * samples[:, 2] + samples[:, 3] + 50 < 0]
        X_class2.extend(filtered_samples)

    X_class2 = np.array(X_class2[:100])

    # Combine the data
    X_train = np.vstack((X_class1, X_class2))
    y_train = np.hstack((np.ones(len(X_class1)), -1 * np.ones(len(X_class2))))


    # Generate test data (similar to training data)
    #New random seed for different data
    np.random.seed(25)
    X_class1test = []
    while len(X_class1test) < 100:
        samples = np.random.uniform(-25, 25, (100, 4))
        # Class 1: {(x1, x2, x3, x4) | 0.5x1    x2    10x3 + x4 + 50 > 0}
        filtered_samples = samples[0.5 * samples[:, 0] - samples[:, 1] - 10 * samples[:, 2] + samples[:, 3] + 50 > 0]
        X_class1test.extend(filtered_samples)

    X_class1test = np.array(X_class1[:100])

    # Generate 100 samples for Class 2
    X_class2test = []
    while len(X_class2test) < 100:
        samples = np.random.uniform(-25, 25, (100, 4))
        # Class 2: {(x1, x2, x3, x4) | 0.5x1    x2    10x3 + x4 + 50 < 0}
        filtered_samples = samples[0.5 * samples[:, 0] - samples[:, 1] - 10 * samples[:, 2] + samples[:, 3] + 50 < 0]
        X_class2test.extend(filtered_samples)
```

```python

    X_class2test = np.array(X_class2[:100])

    # Combine the data
    X_test = np.vstack((X_class1test, X_class2test))
    y_test = np.hstack((np.ones(len(X_class1)), -1 * np.ones(len
        (X_class2))))

    return X_train, y_train, X_test, y_test
```

Listing 2: `ClassGen.py`

```python
import numpy as np
from perceptron import Perceptron
from plot import plot_and_draw
from ClassGen import generate_case1_data, generate_case2_data,
    generate_case3_data

def get_data(case):
    """
    Retrieve the appropriate dataset based on the case number.
    """
    data_generators = {
        '1': generate_case1_data,
        '2': generate_case2_data,
        '3': generate_case3_data
    }

    # Return the dataset based on the case, assuming valid input
        is provided
    return data_generators[case]()

def train_and_evaluate(case, use_gd, learning_rate):
    print()
    """
    Train a Perceptron classifier and evaluate its performance.
    """
    X_train, y_train, X_test, y_test = get_data(case)
    n_features = X_train.shape[1]
```

```python
    perceptron = Perceptron(n_features, learning_rate)

    if use_gd:
        print(f"Using Gradient Descent learning with a learning rate of {learning_rate}")
        perceptron.fit_GD(X_train, y_train)
    else:
        print(f"Using Rosenblatt [1958] learning with a learning rate of {learning_rate}")
        perceptron.fit(X_train, y_train)

    y_pred = perceptron.predict(X_test)
    misclassified = np.sum(y_pred != y_test)
    accuracy = (len(y_test) - misclassified) / len(y_test) * 100

    print(f"Misclassified samples: {misclassified}")
    print(f"Accuracy: {accuracy:.2f}% \n")

    if int(case) != 3:
        plot_and_draw(X_test, y_test, y_pred, perceptron, case)
```

Listing 3: `trainer.py`

```python
from trainer import train_and_evaluate

def prompt_user_input():
    """
    Prompt the user for input regarding the case, learning algorithm, and learning rate.
    Returns a tuple containing (case, use_gd, learning_rate).
    """
    case = input("Which case would you like to test? (1, 2, or 3): \n")
    use_gd_input = input("Use gradient descent? (Y/N): \n").strip().upper()
    learning_rate = float(input("Please select a learning rate: (0.001 - 0.1): \n"))

    # Validate the use_gd input
```

```
    use_gd = use_gd_input == 'Y'

    return case, use_gd, learning_rate

def validate_case(case):
    """
    Validate if the input case is one of the expected values ('1', '2', '3').
    """
    valid_cases = {'1', '2', '3'}
    return case in valid_cases

def main():
    """
    Main function to prompt user input, validate it, and train/evaluate the model.
    """
    case, use_gd, learning_rate = prompt_user_input()

    # Validate the case and proceed with training and evaluation
    if validate_case(case):
        train_and_evaluate(case, use_gd, learning_rate)
    else:
        print("Invalid input. Please enter 1, 2, or 3.")

if __name__ == "__main__":
    main()
```

Listing 4: `main.py`